The C Cyclic Executive Framework

By: Nicholas Khorasani

Introduction:

What is The C Cyclic Executive Framework?

The C Cyclic Executive Framework (CCEF) is a minimal C framework designed to make developing a Cyclic Executive scheduler for an embedded system as simple as possible.

Why should I use The C Cyclic Executive Framework?

A cyclic executive scheduler is a great choice for many embedded systems. Cyclic executive schedules are simple, deterministic and, therefore, highly predictable.This makes cyclic executives an especially good choice for safety-critical embedded systems where failure patterns must be known. Though cyclic executive schedulers are relatively simple, there is a large amount of overhead required for implementing one. However, with the C Cyclic Executive Framework, developing and running a cyclic executive schedule has never been easier. CCEF handles all the scheduling under the hood so all you need to do is define the schedule with the easy-to-understand API.

What does The Simple C FSM Framework include?

The framework consists of four files: scheduler.h, scheduler.c, JobQueue.h and JobQueue.c. These files include all the structs and functions you need in order to develop and run a cyclic executive schedule. Please refer to the **Getting Started** section for details on how to use The C Cyclic Executive Framework.

Getting Started:

Installation:

To install and use The C Cyclic Executive Framework, follow the steps below.

Step 1: Download the headers (scheduler.h & JobQueue.h) and C files (scheduler.c & JobQueue.c) to your development machine.

Step 2: Add all the files to your project's source code.

Step 3: include the scheduler.h header file in the file you wish to create and run your cyclic executive schedule (see code sample below).

```
#include "scheduler.h"
```

How To Create and Run a Cyclic Executive Schedule Using The CCEF:

To create and run an cyclic executive schedule using CCEF you will need to do the following:

1. Define your schedule's tasks by creating Task and TaskSimplified structs.
2. Define your frames by creating a FrameTemplate array.
3. Initialize a CyclicExecutiveSchedule struct variable.
4. Call the startScheduler(CyclicExecutiveSchedule* schedule) function.

Important Note: Before continuing to Step 1 make sure you have done the following:

1. You have written out a function for each of your tasks.
2. You have calculated each task's worst case execution time of each of your tasks.
3. You have created a feasible cyclic executive schedule for your tasks.

Note: If you do not know what a cyclic executive schedule is or how to one see the Example Cyclic Executive Schedule For a Turbofan Jet Engine Controller section.

1: Defining Your Schedule's Tasks:

To define your scheduled tasks you need Task and TaskSimplified structs(see code below for the typedef of the structs). To do this you need to define taskIDs for all your tasks and then initiliaze a Task variable and a TaskSimplified variable for each task.

```
typedef struct {
   int taskID;                  // A unique integer value (must start at 0)
   TaskType taskType;           // The Type of the task (PERIODIC, SPORADIC or APERIODIC)
   int WCET;                    // The Worst Case Execution time of the jobFunction (microseconds)
   int relativeDeadline;        // The time a job of this task must complete in (microseconds)
   int period;                  // Only for Periodic Tasks (microseconds)
   int minimumInterArrivalTime; // Only for sporadic Tasks (microseconds)
   void (*jobFunction)(void);   // The task function
} Task;

typedef struct {
   int taskID; // Either the taskID from the task struct or SLACK_TASK_ID
   int length; // This is WCET (microseconds) for the tasks you defined or length (microseconds) for a slack
} TaskSimplified;
```

Defining taskIds:

Before you define your Task structs you need to define the taskIDs of each of your tasks. Your taskIDs must start at 0 and increase by 1 for each task you have. The example below shows how your taskIDs should be defined (feel free to choose whatever names you want)!

```
#define TASK_ZERO_ID   0
#define TASK_ONE_ID    1
#define TASK_TWO_ID    2
#define TASK_THREE_ID  3
#define TASK_FOUR_ID   4
```

Note: If you were to add a sixth task the taskID would have to be 5.

Creating Task Structs:

Now that you have created your taskIDs it is time to define a Task struct variable for each of the tasks. The CCEF supports periodic tasks, sporadic tasks and aperiodic tasks. The nuances of defining each of these task types is detailed below.

Note: Depending on the type of the task, certain fields do not need to be defined (e.g when defining periodic tasks you do not need define a minimum interarrival time). In these cases it is recommended that you use the ARG_NOT_NEEDED macro defined in the scheduler.h file.

Defining Periodic Task Struct Variables:

To define a periodic task you need to initialize a Task struct variable with taskType “PERIODIC”. For periodic tasks you do not need to define the minimumInterArrivalTime field. Please note that **all time fields must be specified in microseconds (µs)**.

```
Task Task_One = {
  TASK_ONE_ID,          // The ID of your task
  PERIODIC,             // Must be PERIODIC for periodic tasks
  T1_WCET,              // The Worst Case Execution time of the jobFunction (µs)
  T1_REL_DEADLINE,      // Relative Deadline of a job of this task (µs)
  T1_PERIOD,            // Period of Task (µs)
  ARG_NOT_NEEDED,       // Minimum Inter Arrival Time is not needed for periodic tasks
  engine_low_pressure_compression_shaft_task // Task Function
}
```

Defining Sporadic Task Struct Variables:

To define a sporadic task you need to initialize a Task struct variable with taskType “SPORADIC”. For sporadic tasks you do not need to define the period field.

```
Task Task_Two = {
  TASK_TWO_ID,            // The ID of your task
  SPORADIC,               // Must be SPORADIC for periodic tasks
  T2_WCET,                // The Worst Case Execution time of this tasks jobFunction (µs)
  T2_REL_DEADLINE,        // Relative Deadline of a job of this task (µs)
  ARG_NOT_NEEDED,         // Period is not needed for sporadic tasks
  T2_INTER_ARRIVAL_TIME,  // Minimum Inter Arrival Time of the sporadic task (µs)
  engine_low_pressure_compression_shaft_task // Task Function
}
```

Defining Aperiodic Task Struct Variables:

To define an aperiodic task you need to initialize a Task struct variable with taskType “APERIODIC”. For aperiodic tasks you do not need to define the period and minimumInterArrivalTime fields.

```
Task Task_Three = {
  TASK_THREE_ID,       // The ID of your task
  APERIODIC,           // Must be APERIODIC for periodic tasks
  T3_WCET,             // The Worst Case Execution time of this tasks jobFunction (µs)
  T3_REL_DEADLINE,     // Relative Deadline of a job of this task (µs)
  ARG_NOT_NEEDED,      // Period is not needed for aperiodic tasks
  ARG_NOT_NEEDED,      // Minimum Inter Arrival Time is not needed for aperiodic tasks
  engine_low_pressure_compression_shaft_task // Task Function
}
```

Creating TaskSimplified Structs:

Once you have created a Task struct variable for each of your tasks you must create a TaskSimplified struct variable for each of your tasks. You will use these struct variables in Step 2 to define your schedules frames.

Defining a TaskSimplified Struct is very simple as it only contains two fields. The taskID of a task and the WCET of the task. These should be the same as their Task struct counterpart. Below are examples of how to define TaskSimplified struct variables.

```
TaskSimplified Task_One_Simplified =   { TASK_ONE_ID, T1_WCET }
TaskSimplified Task_Two_Simplified =   { TASK_TWO_ID, T2_WCET }
TaskSimplified Task_Three_Simplified = { TASK_THREE_ID, T3_WCET }
```

Note: There is no difference in how you define TaskSimplified struct variables for periodic, sporadic and aperiodic tasks.

2: Creating a FrameTemplate Array to Define Your Schedules Frames:

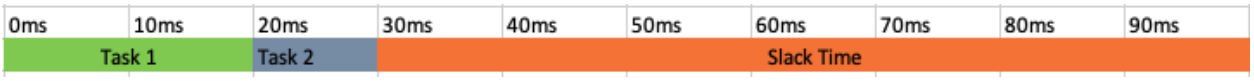
After you have created all your Task and TaskSimplified struct variables it is now time to define your schedule using the TaskSimplified struct variables you created in Step 1. However, before you can define your whole schedule you first need to define each of your frames individually.

First, you need to define an ID for each of your frames. These IDs should start at 0 for the first frame in your schedule and increment by one for each additional frame. Here is an example of how your frame ID definitions might look if you have 5 frames in your schedule:

```
#define FRAME_ZERO_ID 0
#define FRAME_ONE_ID 1
#define FRAME_TWO_ID 2
#define FRAME_THREE_ID 3
#define FRAME_FOUR_ID 4
```

Secondly, *you need to make an array of TaskSimplified struct variables for each of the frames in your schedule*. Each of these arrays will represent one frame of your schedule. The first element of this array is the first task that runs in this frame, the second element runs second and so on. If you have slack time in your frame you should create a TaskSimplified struct variable (using the SLACK_TASK_ID macro defined in schedule.h) and insert it in its appropriate location.

Here is an example of a 100ms frame where Task 1 has a WCET of 20ms, Task 2 has a WCET of 10ms and the rest of the frame (70ms) is allocated as slack time.



And this is how you would define the frame above in the C Cyclic Executive Framework:

```
// See Step 1 to see how to create TaskSimplified struct variables
TaskSimplified Task_One_Simplified = {TASK_ONE_ID, 20000} // periodic task
TaskSimplified Task_Two_Simplified = {TASK_TWO_ID, 10000} // periodic task

TaskSimplified frame_one_schedule[3] {
   Task_One_Simplified,
   Task_Two_Simplified,
   {SLACK_TASK_ID, 70000} // represents 70ms of slack time at the end of the frame
} State;
```

Note-1: the sum of the WCET of your tasks and the length of the slack time in your frame needs to add up to your frame length.

Note-2: Only your periodic tasks should appear in the TaskSimplified array that represents a frame. Your aperiodic and sporadic tasks will be scheduled into slack time when they are released. See the section on Releasing Sporadic and Aperiodic Tasks for more detail.

Finally, you need to create an array of type FrameTemplate. Which represents the periodic tasks and slack time of each frame in your schedule. The FrameTemplate struct has 3 fields: int FrameID, TaskSimplified* frame (the array of TaskSimplified variables you made earlier in Step 2), and int numTasks (the number of jobs in the TaskSimplified array 'frame').

Here is an example of how to create a FrameTemplate array:

```
// Initialize FrameTemplate Array
static FrameTemplate frame_template_array[NUMBER_OF_FRAMES_IN_SCHEDULE] = {
  {FRAME_ZERO_ID, frame_zero_schedule, LENGTH_OF_FRAME_ZERO_SCHEDULE_ARRAY},
  {FRAME_ONE_ID, frame_one_schedule, LENGTH_OF_FRAME_ONE_SCHEDULE_ARRAY},
  {FRAME_TWO_ID, frame_two_schedule, LENGTH_OF_FRAME_TWO_SCHEDULE_ARRAY},
  {FRAME_THREE_ID, frame_three_schedule, LENGTH_OF_FRAME_THREE_SCHEDULE_ARRAY},
  ... //
  ... // Add as many frames as you want!
  ... //
};
```

3: Initialize a CyclicExecutiveSchedule struct variable:

Once you have created the FrameTemplate array that represents your whole cyclic executive schedule it is time to create a Task array, create a CyclicExecutiveSchedule struct variable and initialize all the fields to the appropriate values.

Before we can do that we first need to make an array of the Task struct variables we created in Step 1. Here is an example of how to do that.

```
// Initialize Task Array
static Task tasks[NUMBER_OF_TASKS] = {
  Task_One,
  Task_TWO,
  Task_Three,
  ... //
  ... // Add all the tasks you created in Step 1!
  ... //
};
```

Now we are ready to create and initialize a CyclicExecutiveSchedule struct variable. Here is an example of how this is done:

```
CyclicExecutiveSchedule schedule;
schedule.frameTemplates = frame_template_array; // Created in Step 2!
schedule.numFrames = NUMBER_OF_FRAMES_IN_SCHEDULE; // Length of frame_template_array
schedule.frameLengthMicroSeconds = FRAME_LENGTH; // The length of your frames (µs)
schedule.tasks = tasks; // Created above in Step 3!
schedule.numTasks = NUM_TASKS;  // The number of tasks (the length of the tasks variable)
```

Your cyclic executive schedule is now fully defined! Please proceed to Step 4 to learn how to run it.

4: Call the startScheduler(CyclicExecutiveSchedule* schedule) function:

Now that you have created and initialized a CyclicExecutiveSchedule struct variable it is time to call your cyclic executive schedule!

To do this all you need to do is pass the address of the CyclicExecutiveSchedule struct variable to the startScheduler() function. Below is an example of how to do that:

```
startSchedule(&schedule);
```

Congratulations! You have now successfully defined and run a cyclic executive schedule using The C Cyclic Executive Framework!

Releasing Sporadic and Aperiodic Tasks:

Many cyclic executive schedules include lots of slack time where sporadic and aperiodic jobs can be scheduled. In order for a sporadic or aperiodic job to be scheduled you just need to add it to the appropriate job queue. The CyclicExecutiveSchedule struct contains the fields: JobQueue sporadicJobs and JobQueue aperiodicJobs. If you would like to add a sporadic job you need to add it to the sporadicJobs queue and if you would like to release an aperiodic job you need to add it to the aperiodicJobs queue. Details on how to add jobs to these queues are provided in the steps below.

Whenever you want to release a sporadic or aperiodic job (in an interrupt handler or some other location) you need to follow these steps:

1. Get a context for the sporadic or aperiodic job to run in by calling the getNextContext() function.
2. Calculate the deadline of the job using the timer_read function and the relative deadline of the task.
3. Call the insertJobIntoQueue() function with the appropriate arguments.
    a. The arguments you need to provide are:
        i. The address of the job queue you wish to add a job too (sporadicJobs or aperiodicJobs)
        ii. The taskType of the new job

iii. The worst case execution time of the job
iv. The jobFunction for the new job
v. The Context to run the job in
vi. **Note: for the last two arguments of the function just pass in '0'**

Here is an example of how to release a sporadic job:

```
// Get the Task variable of the sporadic or aperiodic job you want to release
Task sporadicTask = tasks[SPORADIC_JOB_ID];
// Step 1: Get a context to run the job in
int context = getNextContext();
// Step 2: Calculate the deadline of the job
int deadline = timer_read() + sporadicTask.relativeDeadline;
// Step 3: Call the insertJobIntoQueue function
insertJobIntoQueue(&(schedule.sporadicJobs), sporadicTask.taskType, sporadicTask.WCET,
deadline, sporadicTask.jobFunction, context, 0, 0);
```

Here is an example of how to release an aperiodic job:

```
// Get the Task variable of the aperiodic job you want to release
Task aperiodicTask = tasks[APERIODIC_JOB_ID];
// Step 1: Get a context to run the job in
int context = getNextContext();
// Step 2: Calculate the deadline of the job
int deadline = timer_read() + aperiodicTask.relativeDeadline;
// Step 3: Call the insertJobIntoQueue function
insertJobIntoQueue(&(schedule.aperiodicJobs), aperiodicTask.taskType, aperiodicTask.WCET,
deadline, aperiodicTask.jobFunction, context, 0, 0);
```

And that's it! Once you add a sporadic or aperiodic job to the appropriate JobQueue The C Cyclic Executive Framework will schedule the job in the next slack time the job fits in!

Note: The C Cyclic Executive Framework prioritizes sporadic jobs over aperiodic jobs. So if a sporadic and aperiodic job is released at the same time the sporadic job will be scheduled first.

A Cyclic Executive Schedule For a Turbofan Jet Engine Controller:

In this embedded system there are 5 tasks (3 periodic, 1 sporadic, 1 aperiodic). The details of the tasks are shown in Table 1 below.

Task	Task Type	Phase (ms)	Worst Case Execution Time (ms)	Period (ms)	Relative Deadline (ms)	Minimum Interarrival Time (ms)
Low pressure compression shaft monitoring	Periodic	0	20	600	600	N/A
High pressure compression shaft monitoring	Periodic	100	20	300	300	N/A
Fuel injection control	Periodic	0	10	100	150	N/A
Airflow monitoring	Sporadic	N/A	50	N/A	200	400
Vibration analysis	Aperiodic	N/A	30	N/A	500	N/A

Table 1: All relevant information for each of the Tasks in the Turbofan Jet Engine Controller embedded system.

To create a cyclic executive schedule we need to follow these steps:

1. Find the hyperperiod of the periodic tasks
2. Chose a frame time that satisfies the following three criteria:
    a. Criteria 1: Frame time should be larger than the maximum execution time of all jobs
    b. Criteria 2: Frame time should evenly divide the hyperperiod
    c. Criteria 3: Frame should be small enough that between the release time and deadline of every job there is at least one frame (i.e 2f - gcd(Pi, f) <= Di where f = frame time, Pi = period of task i and Di = deadline of task i)
3. Create the frames!

Using these criteria let's make a cyclic executive schedule for our Turbofan Engine Controller embedded system.

Step 1: Find Hyper Period of The Periodic Tasks

The hyperperiod for these tasks is 600 ms since it is the smallest number that 600, 300 and 100 divide evenly.

Step 2: Choose a Frame Time That Satisfies Criteria 1, 2, & 3

From Criteria 1 we know that our frame time must be greater than or equal to 50ms so that no task is sliced.

From Criteria 2 we know that our frame time must evenly divide 600. Therefore, we have the options {50ms, 60ms, 75ms, 100ms, 120ms, 150ms, 200ms, 300ms, and 600ms}.

From Criteria 3 we know that there should be at least 1 frame between the release time and deadline of each job. Let's pick a frame size of 100ms and see if it satisfies the criteria 2f - gcd(Pi, f) <= Di.

For the Fuel injection control task we see that 100 ms passes the criteria since 2*100 = gcd(100, 100) <= 150 gives 100 <= 150 which is true!

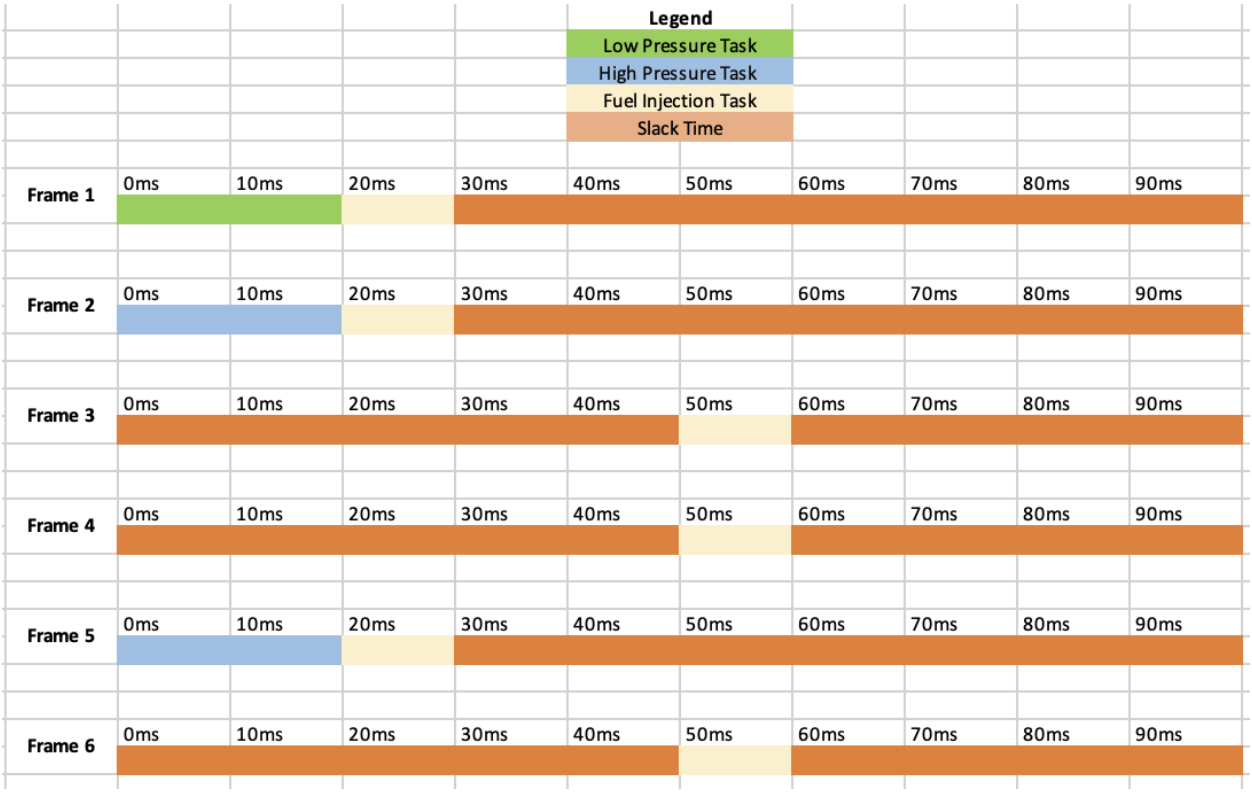
Additionally, we know that the criteria will be satisfied for the other two periodic tasks (Low and High Pressure Compression Shaft Monitoring tasks) since they have a longer period and the gcd will also be 100.

Now that we know 100ms is an appropriate frame time we can make our frames!
Note: 100ms was also chosen due to its ability to fit both the sporadic and aperiodic tasks in the slack time of some of the frames as you will see below.

Creating the Frames:

Now that we have selected a frame time of 100ms we can begin to construct our frames.

Here is one of the possible frame configurations and the reasons behind the choices made here.



Reasons for this configuration:

Frame 1 schedules Low Pressure Task, then Fuel injection right after it. This is so if any sporadic or aperiodic times come they will be able to fit into the 70ms of slack time. Frames 2 and 5 were designed with the same idea.

Frames 3, 4 & 5 have 50ms of slack time followed by the Fuel Injection task followed by another 40 ms of slack time. This is to guarantee that either the sporadic task (50ms WCET) or aperiodic tasks (30ms WCET) or - both of them - will be able to be scheduled in this frame (this is also why the frame length was made to be 100ms).